Take-home Exam in Advanced Programming

Deadline: Thursday, November 7, 16:00

Version 1.1

Preamble

This is the exam set for the course Advanced Programming, B1-2013. This document consists of 15 pages; make sure you have them all. The set consists of 3 questions. Your solution will be graded as a whole.

In the event of errors or ambiguities in the exam text, you are expected to state your assumptions as to the intended meaning. You may ask for clarifications in the discussion forum, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 5–10 pages, not counting appendices, presenting your solutions and reflections. The report should contain all your source code in appendices. The report must be a PDF.
- *The source code* should be in a zip-file containing one directory called `src` (which may contain further subdirectories).

Make sure that you follow the format specifications (PDF and zip). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the course web page on Absalon.

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.

- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.
- It is important that you implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some working code for all questions.

Exam Fraud

The exam is an individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course.

You are only allowed to discuss how a question is to be interpreted with the teachers and their assitants on the discussion forum set up on the course web page on Absalon. If you are afraid that your question does not fall into this category, you can instead send an email instead.

Specifically, but not exclusively, you are **not** allowed to discuss any part of the exam with any other student nor to copy parts of other students' programs. Submitting answers you have not written yourself or *sharing your code with others* is considered exam fraud. Likewise, make sure that you use proper academic citation for the material you use (including what you may find on the Internet, e.g., pieces of code). Also note, that these rules mean that it is not allowed to copy any part of the exam and make it online in other forums (IRC, exam banks, chatrooms, or suchlike) than the discussion forum.

During the exam period, students are not allowed to answer questions, *only teachers and their assitants are allowed to answer questions* on the discussion forum.

Emergency Webpage

There is an emergency web page at

`http://www.diku.dk/~kflarsen/ap-e2013/`

in case Absalon becomes unstable. The page will describe what to do if Absalon becomes unreachable during the exam, especially what to at the hand-in deadline.

Question 1: The Salsa Language

Salsa is a domain specific language for specifying animations, roughly based on the ideas from POLKA[1] and Samba[2]. You can find some example Salsa programs in Appendix A and B.

In this question you must use one of the three monadic parser libraries, `SimpleParse.hs`, `ReadP` or Parsec, from the course to write a parser for Salsa. The parser must be implemented in a `SalsaParser` module. You find Haskell skeletons for the parser and abstract syntax tree at Absalon.

If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators`), in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

[1]See also `http://www.cc.gatech.edu/gvu/ii/softvis/parviz/polka.html`.
[2]See also `http://www.cc.gatech.edu/gvu/ii/softvis/algoanim/samba.html`.

Grammar

Your parser must accept programs written in the following grammar:

Program	::=	*DefComs*
DefComs	::=	*DefCom*
	|	*DefCom DefComs*
DefCom	::=	*Command*
	|	*Definition*
Definition	::=	`'viewdef'` *VIdent Expr Expr*
	|	`'rectangle'` *SIdent Expr Expr Expr Expr Colour*
	|	`'circle'` *SIdent Expr Expr Expr Colour*
	|	`'view'` *VIdent*
	|	`'group'` *VIdent* `'['` *VIdents* `']'`
Command	::=	*SIdents* `'->'` *Pos*
	|	*Command* `'@'` *VIdent*
	|	*Command* `'||'` *Command*
	|	`'{'` *Command* `'}'`
VIdents	::=	*VIdent*
	|	*VIdent VIdents*
SIdents	::=	*SIdent*
	|	*SIdent SIdents*
Pos	::=	`'('` *Expr* `','` *Expr* `')'`
	|	`'+'` `'('` *Expr* `','` *Expr* `')'`
Expr	::=	*Prim*
	|	*Expr* `'+'` *Prim*
	|	*Expr* `'-'` *Prim*
Prim	::=	*integer*
	|	*SIdent* `'.'` `'x'`
	|	*SIdent* `'.'` `'y'`
	|	`'('` *Expr* `')'`
Colour	::=	`'blue'` | `'plum'` | `'red'` | `'green'` | `'orange'`

Where `@` (pronounced "at") has higher precedence than `||` (pronounced "par") and:

- *integer* is a (non-negative) arbitrary-precision integer constant;
- *VIdent* is a nonempty sequence of letters, digits, and underscores (_), starting with an uppercase letter.
- *SIdent* is a nonempty sequence of letters, digits, and underscores (_), starting with a lowercase letter. An *SIdent* may not be one of the reserved words (`viewdef`, `rectangle`, `circle`, `group`, `view`, nor one of the colour names).

Alpha-numeric tokens (*VIdent*s, *SIdent*s, and reserved words) are separated by at least one whitespace (spaces, tabs, and newlines), symbolic tokens are separated by arbitrary whitespace.

If you make any kind of transformation of the grammar as part of your parser construction, make sure to detail them in your report.

Abstract syntax trees

Your parser must construct abstract syntax trees represented with the following data types. You can find this in `SalsaAst.hs`.

```
type Program = [DefCom]
data DefCom = Def Definition
            | Com Command
            deriving (Show, Eq)
data Definition = Viewdef Ident Expr Expr
                | Rectangle Ident Expr Expr Expr Expr Colour
                | Circle Ident Expr Expr Expr Colour
                | View Ident
                | Group Ident [Ident]
                deriving (Show, Eq)
data Command = Move [Ident] Pos
             | At Command Ident
             | Par Command Command
             deriving (Show, Eq)
data Pos = Abs Expr Expr
         | Rel Expr Expr
         deriving (Show, Eq)
data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Const Integer
          | Xproj Ident
          | Yproj Ident
          deriving (Show, Eq)
data Colour = Blue | Plum | Red | Green | Orange
            deriving (Show, Eq)
type Ident = String
```

The mapping from grammar to constructors should be straightforward, perhaps with the exception absolute positions (x, y), which should use the constructor `Abs`, and relative positions $+(x, y)$, which should use the constructor `Rel`.

Thus, you should implement module `SalsaParser` with the following interface: a function `parseString` for parsing a Salsa program given as a string:

```
parseString :: String -> Either Error Program
```

Where you decide and specify what the type `Error` should be.

Likewise, you should implement a function `parseFile` for parsing a Salsa program given in a file located at the given path:

```
parseFile :: FilePath -> IO (Either Error Program)
```

Where `Error` is the same type as for `parseString`.

You should not change the types for the abstract syntax trees unless there is an update at Absalon telling you explicitly that you can do so.

Together with your parser you must also hand in a test-suite to show that your parser works (or where it does not work).

Question 2: Interpreting Salsa

This question is about writing an interpreter for the Salsa language defined in Question 1. The purpose of Salsa programs is to describe animations in a high-level manner independent of low-level rendering details. Hence, the design enables different graphical backends, more about that in the following. We recommend that you read through the whole question before you start implementing anything, and read the commented examples in Appendix A and B, which both contains abstract syntax trees if your parser is not working.

The key concepts in Salsa are:

- A Salsa *animation* consists of a set of *views* of a given size. Views are rendered as different canvases or windows. Each view has a number of *shapes* (rectangles or circles) on it. (View have traditionally been used for visualising concurrent agents with diverging state.)
- An animation goes through a number of *key frames*, with a number of *intermediate frames* between each key frame. Views are kept in lock-step with respect to frames.
- A Salsa program consists of a sequence of definitions and commands. A command, which may consist of *concurrent* sub-commands, specifies how to go from one key frame to the next, and the interpreter should interpolate the intermediate frames.

  For examples, the Salsa program in Appendix A specifies five key frames, likewise the example program in Appendix B specifies three key frames.

A command or definition a shape is always carried out with respect to the *active view*(s). A view becomes the active view as soon as it is defined. An *Id* can temporarily serve as the active view(s) using the `@` operator. Since we also have a grouping operator, multiple views can serve as the active views. More on this below.

We want to keep the interpreter separate from the graphics engine used. Thus your interpreter will generate low-level instructions for the graphics engine using the following data types, which can be found in the file `Gpx.hs`:

```
type ViewName = String
type ColourName = String
type Frame = [GpxInstr]
type Animation = ([(ViewName, Integer, Integer)], [Frame])
data GpxInstr = DrawRect Integer Integer Integer Integer ViewName ColourName
              | DrawCirc Integer Integer Integer ViewName ColourName
              deriving (Eq, Show)
```

Where a low-level `Animation` consists of two things: a list of all the views defined by the Salsa program, and a list of frames (from first to last frame). Each frame consists of a list graphics instructions for how to draw the frame (on all views), each frame will start blank. The order of instructions does not matter and two frames are considered equal if they contain the same instructions. (However, two frames may be rendered visually different depending on the order of instructions, but that is not your concern, and you are free to exploit this property any way you can.)

Semantics of Salsa

All sizes in Salsa are given in whole pixels. We use the standard graphics style coordinate system, with $(0, 0)$ in the top left corner and the y-coordinates growing down.

- `viewdef` *Id* x y defines a new view of size $x \times y$ pixels and binds it to the identifier *Id*. This definition also sets the active view to *Id*. Both x and y must be non-negative.
- `rectangle` *id* x y w h *col* defines a new rectangle with colour *col*, the lower left corner in (x, y), width w, and height h, and binds the rectangle to the identifier *id* and adds it to the current active view(s). The position of a rectangle is its lower left corner. Both w and h must be non-negative.
- `circle` *id* x y r *col* defines a new circle with colour *col*, the centre in (x, y), and radius r, and binds the circle to the identifier *id* and adds it to the current active view(s). The position of a circle is the centre of the circle. The radius r must be non-negative.
- `view` *Id* sets the active view(s) to *Id*.
- `group` *Id* `[`*Ids*`]` defines the identifier *Id* as a short-hand for referring to all the view identifiers in *Ids*.
- *ids* `->` `(`x`,`y`)` moves each of the shapes bound by *ids* to the absolute position (x, y) on the active view(s).
- *ids* `->` `+(`δx`,`δy`)` moves each of the shapes bound by *ids* to relatively by δx and δy on the active view(s).
- *com* `@` *Id* temporarily makes *Id* the active view(s) for the command *com*.
- *com*$_1$ `||` *com*$_2$ execute *com*$_1$ and *com*$_2$ concurrently until the next key frame. Two concurrent commands must not manipulate the same shape.
- *id*`.x` gives the x-coordinate of the position of *id* (and similar for `.y`). However, the position of a shape is not always uniquely defined (a shape can be at different positions at different views). In case it is not uniquely defined, the position to use is the one with the lowest x-coordinate (y-coordinate for `.y`).

There is a number of ways a Salsa program can fail, for instance: by referring to an undefined identifier, by trying to manipulate a shape on a view where the shape is not defined, by using negative-valued expressions when defining views or shapes, by trying to manipulate the same shape by two concurrent commands, etc.

All of these can be checked statically prior to executing the Salsa program, hence you don't have to worry about proper error handling in this question. If you encounter an error, you should halt the interpreter by calling the built-in Haskell function `error`. Writing the static check, as a separate pass over the abstract syntax tree, is *not* part of the exam.

Your Task

The main objective of this question is that you should demonstrate that you know how to write an interpreter using monads for structuring your code. Thus you should structure your solution along the following lines:

(a) When moving a shape between two key frames, you have to interpolate where it is on the intermediate frames. Shapes move in straight lines, with constant speed, snapping to whole pixels.

Define a function, `interpolate`:

```
type Position = (Integer, Integer)
interpolate :: Integer -> Position -> Position -> [Position]
```

such that, `interpolate` n p_{start} p_{end} computes the list of positions $[p_1,p_2,\ldots,p_{n-1},p_{end}]$ of a shape going from position p_{start} to position p_{end} in n steps. Notice that p_{start} is not part of the list but p_{end} is, hence the resulting list has n elements. We call n the framerate.

(b) During the interpretation of a SALSA program we need to keep track of a context for the commands to be executed in. The context consists of two parts: (1) a read-only environment containing the views, groups, and shapes defined, which of the views are active, and the framerate (to be passed to `interpolate`); and (2) a mutable state giving the position of each shape on the different views. Define a type `Context` to keep track of this context.

(c) Complete the type `SalsaCommand` and make it a monad instance:

```
newtype SalsaCommand a = SalsaCommand {runSC :: Context -> ...}
```

Note that a command cannot change the environment (part (1) of the context above), your type should reflect this.

(d) Define a function `command`, that computes the effect of executing a command, with the type:

```
command :: Command -> SalsaCommand ()
```

(e) Define a type `Salsa`, that is a monad, for building up contexts and for executing commands, so that you can define first the function `definition`

```
definition :: Definition -> Salsa ()
```

And then the function `defCom`

```
defCom :: DefCom -> Salsa()
```

You might want to build a helper function `liftC`

```
liftC :: SalsaCommand() -> Salsa()
```

for commands, remember that a command can't change the environment, but a definition can change the environment.

(f) Define a function `runProg`

```
runProg :: Integer -> Program -> Animation
```

`runProg` n p runs program p with framerate n. That is, with $n - 1$ *intermediate* frames between each key frame. Thus, if n is one, then only the key frames are in the animation. The return value is the animation produced by the program.

Your solution should implemented as module `SalsaInterp` that exports at least `Position`, `interpolate`, and `runProg`.

Advice for your solution

If you have difficulties making your interpreter work for the full Salsa language, then try to make it work for a simpler subset of the language. For instance, by allowing only one view, disallowing concurrent commands, allowing only movement of one shape at a time, or disallowing groups, and so on. If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Question 3: Atomic Transaction Server in Erlang

The task in this question is to implement in Erlang the functionality of a simple server that executes *atomic transactions*. An atomic transaction (the term coming from database systems) is a series of operations performed on some *State* that together appear to be *one* single operation (atomic). Thus, the transaction either results in *all operations being performed without any other operations being performed in-between*, or *none of the operations being performed* (all or nothing).

We encourage that you read and understand the entire question before you start implementing. At the end of the question (page 13) you find a suggested approach of how to get started. At the course page you find two skeleton files (`at_server.erl` and `at_extapi.erl`), which you must use. You can choose to implement the server using OTP or not; if you choose OTP, the return values must match those shown under *Communication primitives* of `at_server.erl` and described below.

The handed-in version of your code should *not* spew unnecessary debug information to the console, and remember also to hand in tests that show correctness or incorrectness of all and parts of your implementation.

A simple atomic transaction server

On a given an atomic transaction server one can always make queries on the current *State* (using the `doquery`-function). But if you want to perform an update on the *State*, you must first begin a transaction (using the `begin_t`-function that creates an intermediate copy of the current state and returns a unique transaction-reference), then perform one or more updates to the intermediate state (with the `update_t`-function) and finally commit the state (using the `commit_t`-function). The result of the commit must either be `ok`, with the side-effect that the *State* of the atomic transaction server is changed to the updated state, or `aborted` in which the *State* is not changed.

There are two ways that an atomic transaction can be aborted:

- An operation on the *State* in the transaction throws an error: this can be both updates (using `update_t`) or queries (using `query_t`). All future synchronous operations on this specific transaction must be answered with `aborted`.
- Another transaction commits its updated *State* before you. Multiple transactions can be started at the same time, but it is only the transaction which commits first that must update the *State*. All other running transactions will then have started from a now obsolete *State* and all currently running and future synchronous operations on these transactions must be answered with `aborted`. The callers of these transaction then have to start a new transaction to update the state. Committing an already successfully committed transaction and performing operations on non-existing transactions both falls in this category.

  This approach a simplified version of what is called *optimistic concurrency control*[3].

[3]In "normal" optimistic concurrency control you would find exactly which parts of the database that have

The Server API

Implement an Erlang module, `at_server`, with the following client API. In the following, *AT* denotes a process-id of *atomic transaction server*, while *Ref* denotes a transaction-id.

There are two functions for creating and stopping a server:

(a) A function `start`(*State*) for creating an atomic transaction server with *State* as the initial state. The function must return `{ok,` *AT*`}`.

(b) A function `stop`(*AT*) for stopping an atomic transaction server; this includes aborting all running transactions. The function must return the current `{ok,` *State*`}` of *AT*.

There is one function for querying the current state:

(c) A function `doquery`(*AT*, *Fun*) that performs a query (no updates) with *Fun* on the current *State* in *AT*. It must be **blocking** and return `{ok,` *Result*`}` if the result of *Fun* (*State*) is *Result* and `error` if the query fails.

There are four functions for working on transactions:

(d) A function `begin_t`(*AT*) for starting a new atomic transaction on *AT*. The function must return a new unique transaction-id as `{ok,` *Ref*`}`. You can use the `make_ref/0` function to produce these references.

(e) A function `query_t`(*AT*, *Ref*, *Fun*) for querying the state using *Fun* in a transaction *Ref* on *AT*. The function must be **blocking** and return `{ok,` *Result*`}` if the result of *Fun* (*State*) is *Result* and `aborted` either if *Fun* fails (in which case this transaction must be aborted) or if the transaction have already been aborted.

(f) A function `update_t`(*AT*, *Ref*, *Fun*) for updating the state using *Fun* (*State*) in a transaction *Ref* on *AT*. The function must **not block** and should return immediately. But if the function *Fun* (*State*) fails, the transaction must be aborted.

(g) A function `commit_t`(*AT*, *Ref*) that, if the transaction *Ref* is _not_ aborted, updates the current state to the state of the transaction, aborts all other running transactions, and returns `ok`. If the transaction is aborted it must return `aborted`.

Your API must be robust against erroneous updating and querying functions. All erroneous behaviour outside a transaction must return `error`, and all erroneous behaviour inside a transaction must return `aborted`.

Concurrent execution of transactions

With the exception of commit, all operations on transactions do not affect other running transactions and the current state, so to exploit this inherent concurrency, each transaction

been updated and then only cancel (role-back) the transactions that touches these parts. Implementing this is outside the scope of this question.

must run in its own process. That is, when an atomic transaction server is asked start a transaction, it must spawn a new "transaction-helper"-process containing its current *State*. When a transaction operation is received by *AT*, it must look up the process in which the transaction operation is to be executed and forward the message to this process.

Note, that your implementation must stop all unused processes and should not keep transaction-helper processes much longer than necessary, especially at commits. Also, processes waiting for answers from aborted transactions must be answered with `aborted` as quickly as possible. **Explain in your report how you solve this problem.**

Extending the API

The API of the atomic transaction is very simple and we would therefore like to extend it with the following four functions. All four functions must only make calls to the server using the API and must be implemented in a module called `at_extapi`.

(h) A function `abort`(*AT*, *Ref*) that forces an abort of the transaction *Ref* on server *AT*.

(i) A function `tryUpdate`(*AT*, *Fun*) that tries to update the *State* on *AT* with the function *Fun*. The function must return one of the three following: `ok` if *Fun* successfully updates the *State* on *AT*, `error` if the function-call *Fun* (*State*) throws an error, and `aborted` if the update failed because other transaction made a commit while running the update function.

(j) A function `ensureUpdate`(*AT*, *Fun*) that ensures the following: either an update on the *State* of *AT* with the function *Fun* is performed, or *Fun* failed on *State*. The function must return `ok` if *Fun* successfully updates the *State* and `error` if the function-call *Fun* (*State*) fails. Thus `aborted` is not an option.

(k) A function `choiceUpdate`(*AT*, *Fun*, *List*) that, given a function *Fun* of two arguments and a list of values *List*, in parallel, tries to update the *State* in *AT* with *Fun* (*State*, *E*) for all elements *E* in *List*. The function call that first successful finishes this update is committed and the result of `choiceUpdate` is the result of this commit.

Suggested approach

It can be advised to start by implementing a version of the atomic transaction server that only runs in a single process. That is, instead of `begin_t` spawning a new process, it copies the current state locally and performs transaction operations on this. When this works, you can extend it to handle transactions in separate processes.

In addition to dividing the implementation into sub-parts, this approach also has the advantage that you can both design tests of the `at_server` and implement the extended API for the single-process server. These are then easy to extend to the full server.

Making a working and tested single-process version of the `at_server` will be considered partly answering of this question. Any additional implementation towards a multi-process server will give more credit. **Write, as part of your assessment, which parts of the server that are working and which are not.**

Appendix A: Salsa program: `simple`

The following Salsa program defines a single view, `Default`, with a single shape, box. The specified animation have five key frames: the box starts at position $(10,400)$, then moves to position $(10,200)$, then moves to position $(110,200)$, then moves to position $(110,400)$, and finally moves back to position $(10,400)$.

Salsa program

```
viewdef Default 400 400
rectangle box 10 400 20 20 green
box -> (10, 200)
box -> +(100, 0)
box -> (110,400)
box -> +(0-100, 0)
```

Abstract syntax

```
[ Def (Viewdef "Default" (Const 400) (Const 400))
, Def (Rectangle "box" (Const 10) (Const 400)
                       (Const 20) (Const 20) Green)
, Com (Move ["box"] (Abs (Const 10) (Const 200)))
, Com (Move ["box"] (Rel (Const 100) (Const 0)))
, Com (Move ["box"] (Abs (Const 110) (Const 400)))
, Com (Move ["box"] (Rel (Minus (Const 0) (Const 100))
                         (Const 0)))]
```

Appendix B: Salsa program: multi

The following Salsa program defines two views, One and Two, with the same two shapes, larry and fawn, on each view, and a view group, Both, for referring to both views. The specified animation have three key frames: larry starts at position (10, 350) and fawn starts at position (300, 350) on both views, then only on view Two larry moves to position (300, 350), at the same time as fawn moves to position (10, 350), and finally on both views all shapes moves up 300 pixels.

Salsa program

```
viewdef One 500 500
viewdef Two 400 400
group Both [One Two]
view Both
rectangle larry 10 350 20 20 blue
rectangle fawn 300 350 15 25 plum

view Two
larry -> (300, 350) || fawn -> (10,350)

view Both
larry fawn -> +(0, 0 - 300)
```

Abstract syntax

```
[ Def (Viewdef "One" (Const 500) (Const 500))
, Def (Viewdef "Two" (Const 400) (Const 400))
, Def (Group "Both" ["One","Two"])
, Def (View "Both")
, Def (Rectangle "larry" (Const 10) (Const 350)
                         (Const 20) (Const 20) Blue)
, Def (Rectangle "fawn" (Const 300) (Const 350)
                        (Const 15) (Const 25) Plum)
, Def (View "Two")
, Com (Par (Move ["larry"] (Abs (Const 300) (Const 350)))
           (Move ["fawn"] (Abs (Const 10) (Const 350))))
, Def (View "Both")
, Com (Move ["larry","fawn"]
            (Rel (Const 0) (Minus (Const 0) (Const 300))))]
```